

Overview of C#

Anders Hejlsberg
Distinguished Engineer
Developer Division
Microsoft Corporation

Overview of C#

Anders Hejlsberg
Distinguished Engineer
Developer Division
Microsoft Corporation

C# – The Big Ideas

- The first component oriented language in the C/C++ family
- Everything really is an object
- Next generation robust and durable software
- Preserving your investment

C# – The Big Ideas

A component oriented language

- C# is the first “component oriented” language in the C/C++ family
- Component concepts are first class:
 - Properties, methods, events
 - Design-time and run-time attributes
 - Integrated documentation using XML
- Enables one-stop programming
 - No header files, IDL, etc.
 - Can be embedded in ASP pages

C# – The Big Ideas

Everything really is an object

- Traditional views

- C++, Java: Primitive types are “magic” and do not interoperate with objects
- Smalltalk, Lisp: Primitive types are objects, but at great performance cost

- C# unifies with no performance cost

- Deep simplicity throughout system

- Improved extensibility and reusability

- New primitive types: Decimal, SQL....
- Collections, etc., work for **all** types

C# – The Big Ideas

Robust and durable software

- Garbage collection
 - No memory leaks and stray pointers
- Exceptions
 - Error handling is not an afterthought
- Type-safety
 - No uninitialized variables, unsafe casts
- Versioning
 - Pervasive versioning considerations in all aspects of language design

C# – The Big Ideas

Preserving your Investment

- **C++ heritage**

- Namespaces, pointers (in unsafe code), unsigned types, etc.
- No unnecessary sacrifices

- **Interoperability**

- What software is increasingly about
- C# talks to XML, SOAP, COM, DLLs, and any .NET Framework language

- **Millions of lines of C# code in .NET**

- Short learning curve
- Increased productivity

Hello World

```
using System;  
  
class Hello  
{  
    static void Main() {  
        Console.WriteLine("Hello world");  
    }  
}
```


C# Program Structure

- **Namespaces**

- Contain types and other namespaces

- **Type declarations**

- Classes, structs, interfaces, enums, and delegates

- **Members**

- Constants, fields, methods, properties, indexers, events, operators, constructors, destructors

- **Organization**

- No header files, code written "in-line"
 - No declaration order dependence

C# Program Structure

```
using System;

namespace System.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }
    }
}
```

Type System

■ Value types

- Directly contain data
- Cannot be null

■ Reference types

- Contain references to objects
- May be null

```
int i = 123;  
String s = "Hello world";
```

i

123

s



"Hello world"

Type System

■ Value types

- Primitives `int i;`
- Enums `enum State { off, on }`
- Structs `struct Point { int x, y; }`

■ Reference types

- Classes `class Foo: Bar, IFoo {...}`
- Interfaces `interface IFoo: IBar {...}`
- Arrays `string[] a = new string[10];`
- Delegates `delegate void Empty();`

Predefined Types

- C# predefined types

- Reference object, string
- Signed sbyte, short, int, long
- Unsigned byte, ushort, uint, ulong
- Character char
- Floating-point float, double, decimal
- Logical bool

- Predefined types are simply aliases for system-provided types

- For example, `int == System.Int32`

Classes

- Single inheritance
- Multiple interface implementation
- Class members
 - Constants, fields, methods, properties, indexers, events, operators, constructors, destructors
 - Static and instance members
 - Nested types
- Member access
 - Public, protected, internal, private

Structs

- Like classes, except
 - Stored in-line, not heap allocated
 - Assignment copies data, not reference
 - No inheritance
- Ideal for light weight objects
 - Complex, point, rectangle, color
 - int, float, double, etc., are all structs
- Benefits
 - No heap allocation, less GC pressure
 - More efficient use of memory

Classes And Structs

```
class CPoint { int x, y; ... }  
struct SPoint { int x, y; ... }
```

```
CPoint cp = new CPoint(10, 20);  
SPoint sp = new SPoint(10, 20);
```



Interfaces

- Multiple inheritance
- Can contain methods, properties, indexers, and events
- Private interface implementations

```
interface IDataBound
{
    void Bind(IDataBinder binder);
}
```

```
class EditBox: Control, IDataBound
{
    void IDataBound.Bind(IDataBinder binder) {...}
}
```

Enums

- Strongly typed
 - No implicit conversions to/from int
 - Operators: +, -, ++, --, &, |, ^, ~
- Can specify underlying type
 - Byte, short, int, long

```
enum Color: byte
{
    Red    = 1,
    Green  = 2,
    Blue   = 4,
    Black  = 0,
    White  = Red | Green | Blue,
}
```

Interfaces

- Multiple inheritance
- Can contain methods, properties, indexers, and events
- Private interface implementations

```
interface IDataBound
{
    void Bind(IDataBinder binder);
}

class EditBox: Control, IDataBound
{
    void IDataBound.Bind(IDataBinder binder) {....}
}
```

Enums

- Strongly typed
 - No implicit conversions to/from int
 - Operators: +, -, ++, --, &, |, ^, ~
- Can specify underlying type
 - Byte, short, int, long

```
enum Color: byte
{
    Red    = 1,
    Green  = 2,
    Blue   = 4,
    Black  = 0,
    white  = Red | Green | Blue,
}
```

Delegates

- Object oriented function pointers
- Multiple receivers
 - Each delegate has an invocation list
 - Thread-safe + and - operations
- Foundation for Framework events

```
delegate void MouseEvent(int x, int y);
```

```
delegate double Func(double x);
```

```
Func func = new Func(Math.Sin);  
double x = func(3.0);
```

Unified Type System

- Everything is an object
 - All types ultimately inherit from object
 - Any piece of data can be stored, transported, and manipulated with no extra work



Unified Type System

■ Boxing

- Allocates box, copies value into it

■ Unboxing

- Checks type of box, copies value out

```
int i = 123;  
object o = i;  
int j = (int)o;
```



Unified Type System

■ Benefits

- ❑ Eliminates “wrapper classes”
- ❑ Collection classes work with all types
- ❑ Replaces OLE Automation's Variant

■ Lots of examples in .NET Framework

```
string s = String.Format(  
    "Your total was {0} on {1}", total, date);
```

```
Hashtable t = new Hashtable();  
t.Add(0, "zero");  
t.Add(1, "one");  
t.Add(2, "two");
```


Component Development

- What defines a component?
 - Properties, methods, events
 - Integrated help and documentation
 - Design-time information
- C# has first class support
 - Not naming patterns, adapters, etc.
 - Not external files
- Components are easy to build and consume

Properties

- Properties are “smart fields”
 - Natural syntax, accessors, inlining

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            caption = value;
            Repaint();
        }
    }
}
```

```
Button b = new Button();
b.caption = "OK";
String s = b.Caption;
```

Indexers

- Indexers are “smart arrays”
 - Can be overloaded

```
public class ListBox: Control
{
    private string[] items;

    public string this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            Repaint();
        }
    }
}
```

```
ListBox listBox = new ListBox();
listBox[0] = "hello";
Console.WriteLine(listBox[0]);
```

Events

Sourcing

- Define the event signature

```
public delegate void EventHandler(object sender, EventArgs e);
```

- Define the event and firing logic

```
public class Button
{
    public event EventHandler click;

    protected void OnClick(EventArgs e) {
        if (click != null) click(this, e);
    }
}
```

Events

Handling

- Define and register event handler

```
public class MyForm: Form
{
    Button okButton;

    public MyForm() {
        okButton = new Button(...);
        okButton.Caption = "OK";
        okButton.Click += new EventHandler(okButtonClick);
    }

    void okButtonClick(object sender, EventArgs e) {
        ShowMessage("You pressed the OK button");
    }
}
```

Attributes

- How do you associate information with types and members?
 - Documentation URL for a class
 - Transaction context for a method
 - XML persistence mapping
- Traditional solutions
 - Add keywords or pragmas to language
 - Use external files, e.g., .IDL, .DEF
- C# solution: Attributes

Attributes

```
public class OrderProcessor
{
    [webmethod]
    public void SubmitOrder(PurchaseOrder order) {...}
}

[XamlRoot("Order", Namespace="urn:acme:b2b-schema.v1")]
public class PurchaseOrder
{
    [XmlElement("shipTo")] public Address shipTo;
    [XmlElement("billTo")] public Address billTo;
    [XmlElement("comment")] public string comment;
    [XmlElement("items")] public Item[] items;
    [XmlAttribute("date")] public DateTime orderDate;
}

public class Address {...}

public class Item {...}
```

Attributes

- **Attributes can be**
 - Attached to types and members
 - Examined at run-time using reflection
- **Completely extensible**
 - Simply a class that inherits from `System.Attribute`
- **Type-safe**
 - Arguments checked at compile-time
- **Extensive use in .NET Frameworks**
 - XML, Web Services, security, serialization, component model, COM and P/Invoke interop, code configuration...

Attributes

```
public class OrderProcessor
{
    [webMethod]
    public void SubmitOrder(PurchaseOrder order) {...}
}

[XmlRoot("Order", Namespace="urn:acme:b2b-schema.v1")]
public class PurchaseOrder
{
    [XmlElement("shipTo")] public Address shipTo;
    [XmlElement("billTo")] public Address billTo;
    [XmlElement("comment")] public string comment;
    [XmlElement("items")] public Item[] items;
    [XmlAttribute("date")] public DateTime orderDate;
}

public class Address {...}

public class Item {...}
```

Attributes

- **Attributes can be**
 - Attached to types and members
 - Examined at run-time using reflection
- **Completely extensible**
 - Simply a class that inherits from `System.Attribute`
- **Type-safe**
 - Arguments checked at compile-time
- **Extensive use in .NET Frameworks**
 - XML, Web Services, security, serialization, component model, COM and P/Invoke interop, code configuration...

Attributes

```
public class OrderProcessor
{
    [webmethod]
    public void SubmitOrder(PurchaseOrder order) {...}
}

[XmlRoot("Order", Namespace="urn:acme:b2b-schema.v1")]
public class PurchaseOrder
{
    [XmlElement("shipTo")] public Address shipTo;
    [XmlElement("billTo")] public Address billTo;
    [XmlElement("comment")] public string comment;
    [XmlElement("items")] public Item[] items;
    [XmlAttribute("date")] public DateTime orderDate;
}

public class Address {...}

public class Item {...}
```

XML Comments

```
class XmlElement
{
    /// <summary>
    ///     Returns the attribute with the given name and
    ///     namespace</summary>
    /// <param name="name">
    ///     The name of the attribute</param>
    /// <param name="ns">
    ///     The namespace of the attribute, or null if
    ///     the attribute has no namespace</param>
    /// <return>
    ///     The attribute value, or null if the attribute
    ///     does not exist</return>
    /// <seealso cref="GetAttr(string)"/>
    ///
    public string GetAttr(string name, string ns) {
        ...
    }
}
```

```

    (0001-9999), MM is the month (01-12), DD is the day (01-31), T is the
    character 'T', hh is the hour (00-23), mm is the minute (00-59), ss
    is the second (00-59), followed, if required, by a decimal point and
    part of up to seven digits.</returns>
    <seealso cref= ToDateTime />

    public static string FromDateTime(DateTime value) {
        return new DateTimeFormatter(27).FormatDateTime(value);
    }

    <summary>
        Converts a date value to a string.</summary>
    <param name= value >
        The date value to convert.</param>
    <returns>
        A string of the form yyyy-MM-DD, where yyyy is the year (0001-9999),
        MM is the month (01-12), and DD is the day (01-31).</returns>
    <seealso cref= ToDate />

    public static string FromDate(DateTime value) {
        return new DateTimeFormatter(10).FormatDate(value);
    }

    <summary>
        Converts a time-of-day value to a string.</summary>
    <param name= value >
        The time-of-day value to convert.</param>
    <returns>
        A string of the form hh:mm:ss.ssssss, where hh is the hour (00-23),
        mm is the minute (00-59), and ss.ssssss is the second (00-59), followed, if
        a decimal point and a fractional part of up to seven digits.</return
    <seealso cref= ToTime />

```

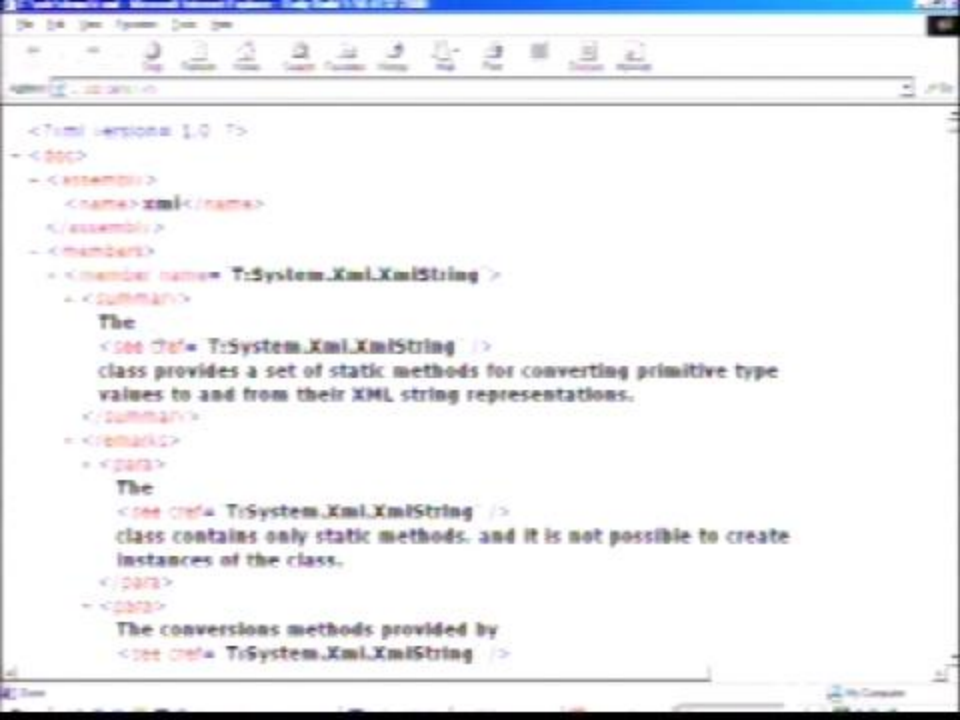
C:\pdc\demo>notepad xml.cs

C:\pdc\demo>csc /doc:t.xml xml.cs

Microsoft (R) C# Compiler Version 7.00.8902 [NGWS runtime 2000.14.1812.7]

Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\pdc\demo>start t.k



1

1

100

1

10

10

10

10

10

10

1

10

10

10

10

10

10

10

```
C:\pdc\demo>notepad xml.cs
```

```
C:\pdc\demo>csc /doc:t.xml xml.cs
```

```
Microsoft (R) C# Compiler Version 7.00.8902 [NGW runtime 2000.14.1812.7]  
Copyright (C) Microsoft Corp 2000. All rights reserved.
```

```
C:\pdc\demo>start t.xml
```

```
C:\pdc\demo>notepad doc.xml
```

```
C:\pdc\demo>
```

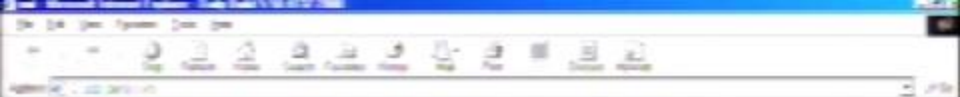
```
C:\pdc\demo>notepad xml.cs
```

```
C:\pdc\demo>csc /doc:t.xml xml.cs
```

```
File Name: Notepad
```

```
C:\pdc\demo>
```

```
<?xml version="1.0"?>
<xml:stylesheet href="doc.xml" type="text/xml"?>
<doc>
  <assembly>
    <name>xml</name>
  </assembly>
  <members>
    <member name="T:System.Xml.XmlString">
      <summary>
        The <see cref="T:System.Xml.XmlString"/> class provides :
        primitive type values to and from their XML string repres
      <remarks>
        <para>The <see cref="T:System.Xml.XmlString"/> class cont
        possible to create instances of the class.</para>
        <para>The conversions methods provided by <see cref="T:S
        given by the XML Schema Specification ( http://www.w3.org
        This in particular means that date and time formatting is
        by the <see cref="T:System.DateTime"/> and <see cref="T:
      </member>
    <member name="M:System.Xml.XmlString.FromBoolean(System.Boolean)
      <summary>
```



System.Xml.XmlString

The [System.Xml.XmlString](#) class provides a set of static methods for converting primitive type values to and from their XML string representations.

Remarks

The [System.Xml.XmlString](#) class contains only static methods, and it is not possible to create instances of the class.

The conversions methods provided by [System.Xml.XmlString](#) follow the conventions given by the XML Schema Specification (<http://www.w3.org/TR/xmlschema-2/>). This in particular means that date and time formatting is different from that provided by the [System.DateTime](#) and [System.TimeSpan](#) types.

FromBoolean(System.Boolean) method

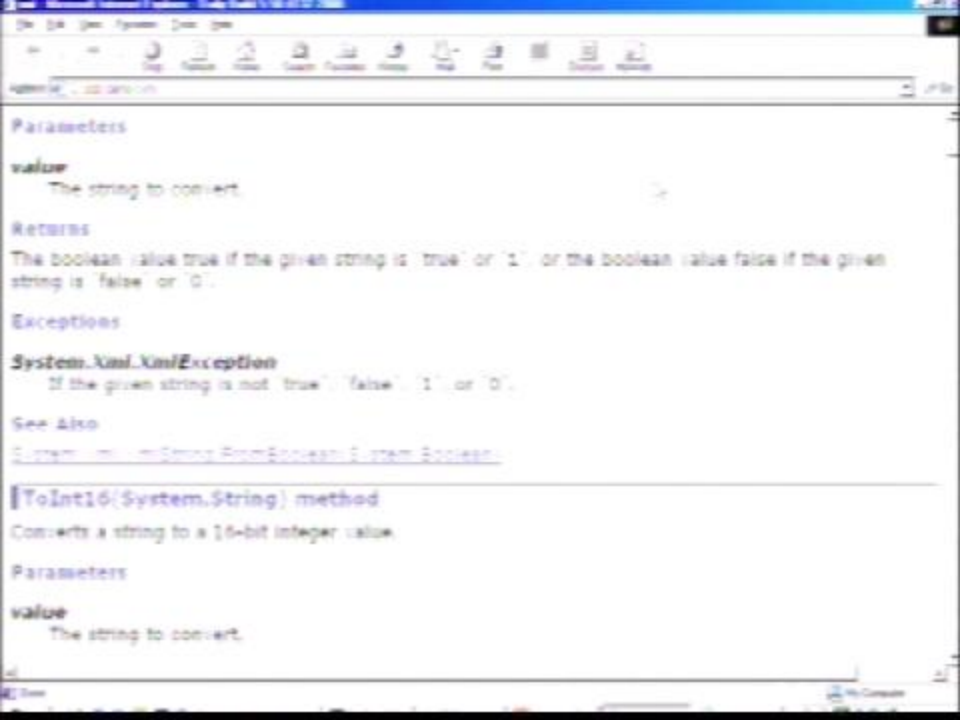
Converts a boolean to a string.

Parameters

value

The boolean value to convert.

Returns



Parameters

value

The string to convert.

Returns

The boolean value `true` if the given string is 'true' or '1', or the boolean value `false` if the given string is 'false' or '0'.

Exceptions

System.Xml.XmlException

If the given string is not 'true', 'false', '1', or '0'.

See Also

[System.Xml.XmlStringFromBoolean\(System.Boolean\)](#)

ToInt16(System.String) method

Converts a string to a 16-bit integer value.

Parameters

value

The string to convert.

Statements And Expressions

- High C++ fidelity
- If, while, do require bool condition
- goto can't jump into blocks
- Switch statement
 - No fall-through, "goto case" or "goto default"
- foreach statement
- Checked and unchecked statements
- Expression statements must do work

```
void Foo() {  
    i = 1;    // error  
}
```

foreach Statement

■ Iteration of arrays

```
public static void Main(string[] args) {  
    foreach (string s in args) Console.WriteLine(s);  
}
```

■ Iteration of user-defined collections

```
foreach (Customer c in customers.OrderBy("name")) {  
    if (c.Orders.Count != 0) {  
        ---  
    }  
}
```

Parameter Arrays

- Can write “printf” style methods
 - Type-safe, unlike C++

```
void printf(string fmt, params object[] args) {  
    foreach (object x in args) {  
        ---  
    }  
}
```

```
printf("%s %i %i", str, int1, int2);
```

```
object[] args = new object[3];  
args[0] = str;  
args[1] = int1;  
args[2] = int2;  
printf("%s %i %i", args);
```


Operator Overloading

- First class user-defined data types
- Used in base class library
 - Decimal, DateTime, TimeSpan
- Used in Frameworks
 - Unit, point, rectangle
- Used in SQL integration
 - SQLString, SQLInt16, SQLInt32, SQLInt64, SQLBool, SQLMoney, SQLNumeric, SQLFloat...

Operator Overloading

```
public struct DBInt
{
    public static readonly DBInt Null = new DBInt();

    private int value;
    private bool defined;

    public bool IsNull { get { return !defined; } }

    public static DBInt operator +(DBInt x, DBInt y) {...}

    public static implicit operator DBInt(int x) {...}
    public static explicit operator int(DBInt x) {...}
}
```

```
DBInt x = 123;
DBInt y = DBInt.Null;
DBInt z = x + y;
```

Versioning

- Overlooked in most languages
 - C++ and Java produce fragile base classes
 - Users unable to express versioning intent
- C# allows intent to be expressed
 - Methods are not virtual by default
 - C# keywords "virtual", "override" and "new" provide context
- C# can't guarantee versioning
 - Can enable (e.g., explicit override)
 - Can encourage (e.g., smart defaults)

Versioning

```
class Base:                                     // version 1
{
}
```

Versioning

```
class Base // version 1
{
}
```

```
class Derived: Base // version 1
{
    public virtual void foo() {
        Console.WriteLine("Derived.foo");
    }
}
```

Versioning

```
class Base // version 2
{
    public virtual void foo() {
        Console.WriteLine("Base.foo");
    }
}
```

```
class Derived: Base // version 1
{
    public virtual void foo() {
        Console.WriteLine("Derived.foo");
    }
}
```

Versioning

```
class Base // version 2
{
    public virtual void foo() {
        Console.WriteLine("Base.foo");
    }
}
```

```
class Derived: Base // version 2a
{
    new public virtual void foo() {
        Console.WriteLine("Derived.foo");
    }
}
```

Versioning

```
class Base // version 2
{
    public virtual void Foo() {
        Console.WriteLine("Base.Foo");
    }
}
```

```
class Derived: Base // version 2b
{
    public override void Foo() {
        base.Foo();
        Console.WriteLine("Derived.Foo");
    }
}
```


Conditional Compilation

- #define, #undef
- #if, #elif, #else, #endif
 - Simple boolean logic
- Conditional methods

```
public class Debug
{
    [Conditional("debug")]
    public static void Assert(bool cond, String s) {
        if (!cond) {
            throw new AssertionError(s);
        }
    }
}
```

Unsafe Code

- COM interop, P/Invoke cover most cases
- Unsafe code
 - Low-level code without leaving the box
 - Enables unsafe casts, pointer arithmetic
- Declarative pinning
 - Fixed statement
- Basically "inline C"

```
unsafe void Foo() {  
    char* buf = stackalloc char[256];  
    for (char* p = buf; p < buf + 256; p++) *p = 0;  
    ...  
}
```

Unsafe Code

```
class FileStream: Stream
{
    int handle;

    public unsafe int Read(byte[] buffer, int index, int count) {
        int n = 0;
        fixed (byte* p = buffer) {
            ReadFile(handle, p + index, count, &n, null);
        }
        return n;
    }

    [DllImport("kernel32", SetLastError=true)]
    static extern unsafe bool ReadFile(int hFile,
        void* lpBuffer, int nBytesToRead,
        int* nBytesRead, overlapped* lpOverlapped);
}
```

Unsafe Code

```
class FileStream: Stream
{
    int handle;

    public unsafe int Read(byte[] buffer, int index, int count) {
        int n = 0;
        fixed (byte* p = buffer) {
            ReadFile(handle, p + index, count, &n, null);
        }
        return n;
    }

    [DllImport("kernel32", SetLastError=true)]
    static extern unsafe bool ReadFile(int hFile,
        void* lpBuffer, int nBytesToRead,
        int* nBytesRead, overlapped* lpOverlapped);
}
```

More Information

<http://msvc/csharp>

- C# Language Reference
- Examples, tutorials, FAQs, etc.

<http://urt>

- Latest .NET SDK builds
- .NET specs, links, etc.

<http://msdn.microsoft.com/events/pdc>

- Slides and info from .NET PDC

http://camweb/Extended_IL/Generics/specs.htm

- Generics in C#